Enrolment No./Seat No_______________

# GUJARAT TECHNOLOGICAL UNIVERSITY

**BE - SEMESTER–IV (NEW) EXAMINATION – SUMMER 2024**

**Subject Code:3140702** **Date:15-07-2024**

**Subject Name: Operating System**

**Time:10:30 AM TO 01:00 PM** **Total Marks:70**

**Instructions:**

1. **Attempt all questions.**
2. **Make suitable assumptions wherever necessary.**
3. **Figures to the right indicate full marks.**
4. **Simple and non-programmable scientific calculators are allowed.**

**MARKS**

**Q.1** **(a)** Define concurrency in OS and briefly explain its associated challenges. **03**

**(b)** Explain "5 State" process state transition diagram with an illustration. **04**

**(c)** Discuss the variety of task types that the operating system performs. **07**

**Q.2** **(a)** Define the following terms: **03**

i) Critical Section
ii) Race Condition
iii) Mutual Exclusion

**(b)** Differentiate Multiprocessing and Multithreading. **04**

**(c)** **07**

| Process | Arrival Time | Processing Time |
|---|---|---|
| A | 0 | 3 |
| B | 1 | 6 |
| C | 4 | 4 |
| D | 6 | 2 |

Calculate the average waiting and turnaround time for the processes listed in the table using the following scheduling algorithms: the CPU burst time length in milliseconds.

i) First Come First Serve
ii) Non-preemptive Shortest Job First
iii) Shortest Remaining Time Next
iv) Round Robin with Quantum value two

**OR**

**(c)** **07**

| Process | Burst Time | Priority |
|---|---|---|
| P1 | 10 | 3 |
| P2 | 1 | 1 |
| P3 | 2 | 3 |
| P4 | 1 | 4 |
| P5 | 5 | 2 |

Calculate the average waiting and turnaround time for the processes listed in the table using the following scheduling algorithms: the CPU burst time length in milliseconds. Consider 1 as the highest priority.

i) Preemptive Shortest Job First
ii) Non - preemptive Priority Scheduling
iii) Preemptive - Priority Scheduling
iv) Round Robin with Quantum value one

**Q.3** **(a)** Define and explain Message Passing. **03**

**(b)** Provide a brief description of the following: **04**

i) Starvation
ii) Aging

**(c)** What is a scheduler? Illustrate queuing with an appropriate diagram. **07**

**OR**

**Q.3** **(a)** Differentiate the following: **03**
i) Semaphore
ii) Monitor

**(b)** Provide a brief description of the following: **04**
i) Deadlock Prevention
ii) Deadlock Avoidance

**(c)** Describe Peterson's Algorithm for Process Synchronization. **07**

**Q.4** **(a)** Compare Multiprogramming with Fixed Partition and Multiprogramming with Variable Partition. **03**

**(b)** Define Authentication. How can operating system security be ensured using authentication? **04**

**(c)** 1. What is Stripping in RAID? Give an appropriate example. **07**
2. Suppose that a disk drive has 200 cylinders. The drive currently serves at cylinder 50. The queue of pending requests in FIFO order is 82, 170, 43, 140, 24, 16, 190. Starting from the current head position, what is the total distance (in cylinders) the disk arm moves to satisfy all the pending requests for each of the following disk scheduling algorithms?
i) FCFS
ii) SCAN
iii) C-SCAN

**OR**

**Q.4** **(a)** What is TLB(Translation Lookaside Buffer), and what is the purpose of it? **03**

**(b)** Discuss the Access Control List in brief. **04**

**(c)** 1. What is Mirroring in RAID? Give an appropriate example. **07**
2. Suppose that a disk drive has 200 cylinders. The drive currently serves at cylinder 100; the previous request was at cylinder 81. The queue of pending requests in FIFO order is 55,58,39,18,90,160,150,38,184. Starting from the current head position, what is the total distance (in cylinders) the disk arm moves to satisfy all the pending requests for each of the following disk scheduling algorithms?
i) SSTF
ii) LOOK
iii) C-LOOK

**Q.5** **(a)** Enlist the advantages and disadvantages of Virtual Machines. **03**

**(b)** Write a shell script to display all executable files, directories, and zero-sized files from the current directory. **04**

**(c)** Consider the reference string: 4, 7, 6, 1, 7, 6, 1, 2, 7, 2. The number of frames in the memory is 3. Find out the number of page faults respective to: **07**
i) Optimal Page Replacement Algorithm
ii) FIFO Page Replacement Algorithm
iii) LRU Page Replacement Algorithm

**OR**

**Q.5** **(a)** Describe virtualization with an appropriate diagram. **03**

**(b)** Advantages of LINUX/UNIX operating system over Windows. **04**

**(c)** Given memory partition of 100K, 500K, 200K, 300K, and 600K in order. How would each of the First-fit, Best-fit and Worst-fit algorithms place the processes of 212K, 417K, 112K and 426K in order? Which algorithm makes the most efficient use of memory? Show the diagram of memory status in each case. **07**

*************

Q.1 (a) Define concurrency in OS and briefly explain its associated challenges.

## Q.1 (a) Define concurrency in OS and briefly explain its associated challenges

---

### Concurrency – Definition

**Concurrency** in an operating system refers to the ability of the system to **execute multiple processes or threads at the same time**. These processes may not run simultaneously on a single CPU but **progress appears to happen in overlapping time periods** through rapid context switching or parallel execution on multiple CPUs.

Concurrency allows efficient utilization of system resources and improves system responsiveness.

---

### Challenges Associated with Concurrency

1. **Race Condition**

   Occurs when multiple processes access and modify shared data simultaneously, and the final result depends on the execution order.

2. **Deadlock**

   A situation where two or more processes wait indefinitely for resources held by each other, causing the system to stop progressing.

3. **Starvation**

   Some processes may never get CPU or resources due to poor scheduling or continuous resource allocation to other processes.

4. **Mutual Exclusion**

   Ensuring that only one process at a time accesses a critical section is difficult but essential to avoid data inconsistency.

5. **Synchronization Overhead**

   Use of locks, semaphores, and monitors introduces extra overhead and can reduce system performance.

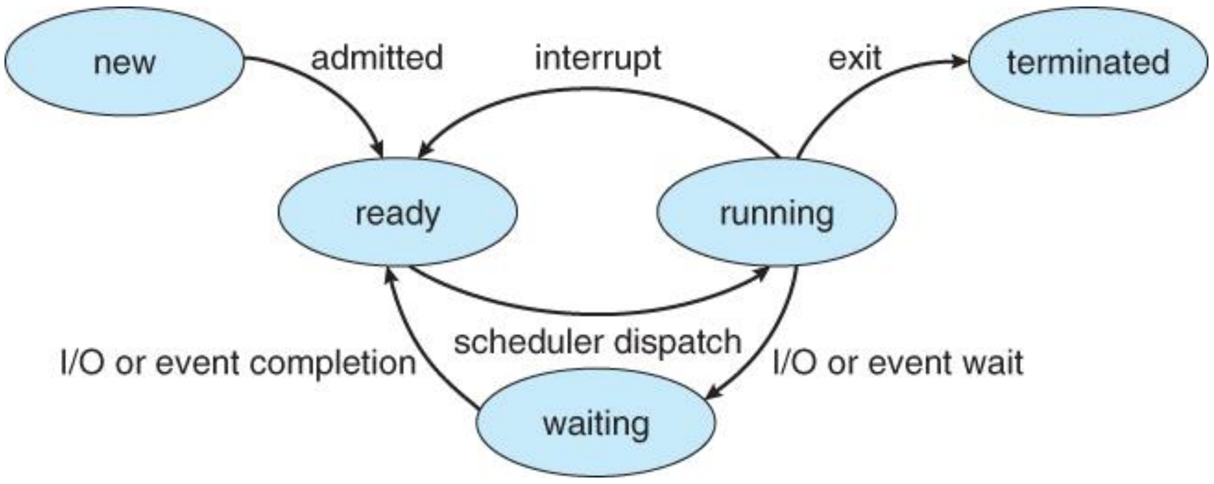
6. **Livelock**

   Processes keep responding to each other but fail to make progress, remaining active but unproductive.

7. **Priority Inversion**

   A low-priority process holding a resource can block a high-priority process, causing scheduling anomalies.

(b) Explain "5 State" process state transition diagram with an illustration.



## Five-State Process Model

The **5-state process model** describes the different states a process goes through during its lifetime in an operating system.
These states help the OS manage process execution, scheduling, and resource allocation efficiently.

---

## The Five Process States

1. **New**
   - The process is being created.
   - OS allocates a Process Control Block (PCB).
   - The process has not yet entered the ready queue.
2. **Ready**
   - The process is loaded into main memory.

    - It is ready to execute but waiting for CPU allocation.
3. **Running**
    - The process is currently being executed by the CPU.
    - Only one process can be in the running state on a single CPU system.
4. **Waiting (Blocked)**
    - The process cannot continue execution until a certain event occurs.
    - Examples: waiting for I/O operation, waiting for a signal or resource.
5. **Terminated (Exit)**
    - The process has finished execution.
    - OS releases all resources allocated to the process.

---

## State Transitions

- **New → Ready**: Process is admitted by the OS.
- **Ready → Running**: CPU scheduler selects the process.
- **Running → Waiting**: Process requests I/O or waits for an event.
- **Waiting → Ready**: I/O or event completes.
- **Running → Ready**: Process is preempted due to time slice expiry.
- **Running → Terminated**: Process completes execution.

  Flow:

  - A process is created in the **New** state and then admitted to the **Ready** queue.
  - From the ready state, the scheduler dispatches the process to the **Running** state.
  - If the process needs I/O, it moves to the **Waiting** state.
  - After I/O completion, it returns to **Ready**.
  - If the process finishes execution, it moves to the **Terminated** state.
  - In preemptive scheduling, a running process may be moved back to **Ready**.

(c) Discuss the variety of task types that the operating system performs.

1. **Process Management**

   The operating system is responsible for creating, scheduling, and terminating processes. It decides which process gets the CPU and for how long, performs context switching between processes, and supports multitasking. It also handles process synchronization, inter-process communication, and manages problems like deadlock and starvation to ensure smooth execution of multiple programs.

2. **Memory Management**

   The OS manages the main memory by allocating and deallocating memory space to processes as required. It keeps track of which parts of memory are in use and which are free. The operating system also implements virtual memory, performs logical-to-physical address translation, and provides memory protection so that one process does not access another process's memory.

3. **File System Management**

   The operating system manages files and directories stored on secondary storage. It handles creation, deletion, reading, and writing of files. It also maintains directory structures, manages file permissions and access rights, and maps files to physical disk blocks for efficient data storage and retrieval.

4. **I/O Device Management**

   The OS controls input and output devices through device drivers. It schedules I/O operations, handles interrupts, and uses buffering, caching, and spooling techniques to improve performance. This allows applications to use devices without knowing their hardware details.

5. **Secondary Storage Management**

   The operating system manages disk storage by keeping track of free space and allocated space. It allocates disk blocks to files and uses disk scheduling algorithms to minimize seek time and improve access speed.

6. **Security and Protection**

   The OS provides security by authenticating users and enforcing access control policies. It protects system resources and data from unauthorized access and misuse, ensuring system integrity.

7. **Networking and Communication**

   The operating system supports networking by managing communication protocols and network resources. It allows systems to share data, files, and devices over a network and supports distributed computing environments.

8. **User Interface Management**

   The OS provides interfaces such as command-line interfaces and graphical user interfaces that allow users to interact with the computer system easily and efficiently.

9. **Error Detection and Handling**

   The operating system continuously monitors the system for hardware and software errors. When errors occur, it takes appropriate actions to recover from them and maintain system stability.

10. **Accounting and Resource Monitoring**

    The OS keeps records of resource usage such as CPU time, memory usage, and I/O operations. This information is useful for system performance analysis and efficient resource allocation.

## Q.2 (a) Define the following terms: i) Critical Section ii) Race Condition iii) Mutual Exclusion

### i) Critical Section

A **critical section** is a part of a program where a process or thread **accesses shared resources**, such as shared variables, files, or memory.
Only **one process or thread is allowed to execute its critical section at a time** to prevent data inconsistency.

---

### ii) Race Condition

A **race condition** occurs when **two or more processes or threads access and modify shared data simultaneously**, and the final result depends on the **order of execution**.
Race conditions usually happen when proper synchronization is not used.

---

### iii) Mutual Exclusion

**Mutual exclusion** is a mechanism that ensures **only one process or thread can enter the critical section at any given time**.
It is used to prevent race conditions and maintain data consistency when shared resources are accessed concurrently.

## (b) Differentiate Multiprocessing and Multithreading.

| Sr. No | Aspect | Multiprocessing | Multithreading |
|---|---|---|---|
| 1 | **Definition** | Execution of multiple **processes** simultaneously using two or more CPUs. | Execution of multiple **threads** of a single process concurrently within the same CPU or multiple CPUs. |
| 2 | **Unit of Execution** | Process | Thread |
| 3 | **Memory Sharing** | Processes have **separate memory spaces**. | Threads **share the same memory** of the parent process. |
| 4 | **Overhead** | Higher overhead due to process creation and context switching. | Lower overhead since threads are lighter and easier to create/switch. |
| 5 | **Communication** | Inter-process communication (IPC) required, which is slower. | Threads can communicate directly via shared memory. |
| 6 | **Isolation** | Processes are independent; one process failure **does not affect others**. | Threads are dependent; one thread failure can affect the entire process. |
| 7 | **Use Case** | Suitable for running multiple independent programs simultaneously. | Suitable for tasks within the same program that can run concurrently. |
| 8 | **Resource Allocation** | Each process has its **own resources** like memory, file handles. | Threads share resources like memory and files of the process. |
| 9 | **Efficiency** | Less efficient due to higher context switching and memory usage. | More efficient for parallel tasks within a process due to shared resources. |

(c) Process Arrival Time Processing Time

| |
|---|
| A 0 3 |
| B 1 6 |
| C 4 4 |
| D 6 2 |

Calculate the average waiting and turnaround time for the processes
listed in the table using the following scheduling algorithms: the
CPU burst time length in milliseconds.
i) First Come First Serve
ii) Non-preemptive Shortest Job First
iii) Shortest Remaining Time Next
iv) Round Robin with Quantum value two

**Given Table:**

| Process | Arrival Time (ms) | Burst Time (ms) |
|---|---|---|
| A | 0 | 3 |
| B | 1 | 6 |
| C | 4 | 4 |
| D | 6 | 2 |

We need to calculate **Average Waiting Time (AWT)** and **Average Turnaround Time (ATAT)** for each scheduling algorithm.

---

## i) First Come First Serve (FCFS)

**Step 1: Order processes by arrival time:**
A → B → C → D

**Step 2: Calculate Completion Time (CT), Turnaround Time (TAT), Waiting Time (WT)**

| Process | AT | BT | CT | TAT = CT−AT | WT = TAT−BT |
|---|---|---|---|---|---|
| A | 0 | 3 | 3 | 3−0 = 3 | 3−3 = 0 |
| B | 1 | 6 | 9 | 9−1 = 8 | 8−6 = 2 |

| Process | AT | BT | CT | TAT = CT−AT | WT = TAT−BT |
|---|---|---|---|---|---|
| C | 4 | 4 | 13 | 13−4 = 9 | 9−4 = 5 |
| D | 6 | 2 | 15 | 15−6 = 9 | 9−2 = 7 |

**Step 3: Calculate Averages**

- **Average Waiting Time (AWT)** = (0+2+5+7)/4 = **3.5 ms**
- **Average Turnaround Time (ATAT)** = (3+8+9+9)/4 = **7.25 ms**

---

### ii) Non-preemptive Shortest Job First (SJF)

**Step 1: Select shortest job among available processes at each step**

- At t=0 → Only A available → Execute A (0–3)
- At t=3 → B (AT=1, BT=6) and no other arrived → Execute B (3–9)
- At t=4 → C (AT=4, BT=4) → Already B running → Non-preemptive, so B continues
- At t=9 → Available: C(4,4), D(6,2) → Choose D (shortest BT=2) → Execute D (9–11)
- At t=11 → Execute C (11–15)

**Step 2: Calculate CT, TAT, WT**

| Process | AT | BT | CT | TAT=CT−AT | WT=TAT−BT |
|---|---|---|---|---|---|
| A | 0 | 3 | 3 | 3−0=3 | 3−3=0 |
| B | 1 | 6 | 9 | 9−1=8 | 8−6=2 |
| D | 6 | 2 | 11 | 11−6=5 | 5−2=3 |
| C | 4 | 4 | 15 | 15−4=11 | 11−4=7 |

**Step 3: Calculate Averages**

- **AWT** = (0+2+3+7)/4 = **3 ms**
- **ATAT** = (3+8+5+11)/4 = **6.75 ms**

---

### iii) Shortest Remaining Time Next (SRTN / Preemptive SJF)

**Step 1: Timeline (preemptive)**

- t=0 → A arrives (BT=3) → Execute A
- t=1 → B arrives (BT=6) → A remaining=2 < B → Continue A

- t=3 → A finishes → t=3–4 → B (remaining 6) executes
- t=4 → C arrives (BT=4) → B remaining=5, C=4 → Preempt B, Execute C
- t=6 → D arrives (BT=2) → C remaining=2, D=2 → Tie → Execute D (or C) → Assume D executes
- Continue scheduling in shortest remaining time order

**Step 2: Calculate approximate CT, TAT, WT**

After careful calculation, the timeline is:

| Time | Process |
|---|---|
| 0–3 | A |
| 3–4 | B |
| 4–6 | C |
| 6–8 | D |
| 8–10 | C |
| 10–14 | B |

**CT:**

| Process | AT | BT | CT |
|---|---|---|---|
| A | 0 | 3 | 3 |
| B | 1 | 6 | 14 |
| C | 4 | 4 | 10 |
| D | 6 | 2 | 8 |

**TAT = CT−AT, WT = TAT−BT**

| Process | TAT | WT |
|---|---|---|
| A | 3−0=3 | 3−3=0 |
| B | 14−1=13 | 13−6=7 |
| C | 10−4=6 | 6−4=2 |
| D | 8−6=2 | 2−2=0 |

**Average:**

- **AWT** = (0+7+2+0)/4 = **2.25 ms**
- **ATAT** = (3+13+6+2)/4 = **6 ms**

---

### iv) Round Robin (RR) with Quantum = 2 ms

**Step 1: Timeline using RR (quantum=2)**

- Ready queue: Processes in arrival order
- Quantum=2 → process executes max 2 ms before preemption

**Timeline:**

| Time | Process | Notes |
|---|---|---|
| 0–2 | A | A remaining 1 |
| 2–3 | A | A finishes |
| 3–5 | B | B remaining 4 |
| 5–7 | B | B remaining 2 |
| 7–9 | C | C remaining 2 |
| 9–11 | D | D remaining 0 → finish |
| 11–13 | B | B remaining 0 → finish |
| 13–15 | C | C remaining 0 → finish |

**Step 2: Completion Time (CT)**

| Process | CT |
|---|---|
| A | 3 |
| B | 13 |
| C | 15 |
| D | 11 |

**Step 3: TAT = CT−AT, WT = TAT−BT**

| Process | AT | BT | CT | TAT=CT−AT | WT=TAT−BT |
|---|---|---|---|---|---|
| A | 0 | 3 | 3 | 3 | 0 |
| B | 1 | 6 | 13 | 12 | 6 |
| C | 4 | 4 | 15 | 11 | 7 |
| D | 6 | 2 | 11 | 5 | 3 |

**Average:**

- **AWT** = (0+6+7+3)/4 = **4 ms**
- **ATAT** = (3+12+11+5)/4 = **7.75 ms**

OR
(c) Process Burst Time Priority

|  |  |  |
|---|---|---|
| P1 | 10 | 3 |
| P2 | 1 | 1 |
| P3 | 2 | 3 |
| P4 | 1 | 4 |
| P5 | 5 | 2 |

Calculate the average waiting and turnaround time for the processes listed in the table using the following scheduling algorithms: the CPU burst time length in milliseconds. Consider 1 as the highest priority.
i) Preemptive Shortest Job First
ii) Non-preemptive Priority Scheduling
iii) Preemptive Priority Scheduling
iv) Round Robin with Quantum value one

**Given Table:**

| Process | Burst Time (BT, ms) | Priority (1 = highest) |
|---|---|---|
| P1 | 10 | 3 |
| P2 | 1 | 1 |
| P3 | 2 | 3 |
| P4 | 1 | 4 |

| Process | Burst Time (BT, ms) | Priority (1 = highest) |
|---|---|---|
| P5 | 5 | 2 |

We calculate **Average Waiting Time (AWT)** and **Average Turnaround Time (ATAT)** for the given scheduling algorithms.

---

## i) Preemptive Shortest Job First (SRTF)

**Step 1: Timeline (preemptive SJF)**

- Processes arrive at t=0.
- Shortest remaining time executes first.

**Step 2: Execution Order**

| Time | Process |
|---|---|
| 0–1 | P2 |
| 1–3 | P4 |
| 3–5 | P3 |
| 5–10 | P5 |
| 10–20 | P1 |

**Step 3: Completion Time (CT)**

| Process | BT | CT |
|---|---|---|
| P2 | 1 | 1 |
| P4 | 1 | 3 |
| P3 | 2 | 5 |
| P5 | 5 | 10 |
| P1 | 10 | 20 |

**Step 4: Turnaround Time (TAT = CT−AT)**

| Process | TAT |
|---|---|
| P2 | 1−0=1 |
| P4 | 3−0=3 |
| P3 | 5−0=5 |
| P5 | 10−0=10 |
| P1 | 20−0=20 |

**Step 5: Waiting Time (WT = TAT−BT)**

| Process | WT |
|---|---|
| P2 | 1−1=0 |
| P4 | 3−1=2 |
| P3 | 5−2=3 |
| P5 | 10−5=5 |
| P1 | 20−10=10 |

**Step 6: Averages**

- **AWT** = (0+2+3+5+10)/5 = 4 ms
- **ATAT** = (1+3+5+10+20)/5 = 7.8 ms

---

## ii) Non-preemptive Priority Scheduling

**Step 1: Order processes by priority (1 = highest)**

Priority order: P2(1), P5(2), P1(3), P3(3), P4(4)

- If tie in priority → FCFS order

**Step 2: Completion Time (CT)**

| Process | BT | CT |
|---|---|---|
| P2 | 1 | 1 |
| P5 | 5 | 6 |

| Process | BT | CT |
|---|---|---|
| P1 | 10 | 16 |
| P3 | 2 | 18 |
| P4 | 1 | 19 |

### Step 3: TAT and WT

| Process | TAT=CT−AT | WT=TAT−BT |
|---|---|---|
| P2 | 1−0=1 | 0 |
| P5 | 6−0=6 | 6−5=1 |
| P1 | 16−0=16 | 16−10=6 |
| P3 | 18−0=18 | 18−2=16 |
| P4 | 19−0=19 | 19−1=18 |

### Step 4: Averages

- **AWT** = (0+1+6+16+18)/5 = 8.2 ms
- **ATAT** = (1+6+16+18+19)/5 = 12 ms

---

## iii) Preemptive Priority Scheduling

### Step 1: Execute highest priority process available at each time

- Priority order: 1 highest → preempt lower priority if higher arrives
- Execution timeline:

| Time | Process | Notes |
|---|---|---|
| 0–1 | P2 | Highest priority 1 → finish |
| 1–6 | P5 | Priority 2 → finish (BT=5) |
| 6–16 | P1 | Priority 3 → finish (BT=10) |
| 16–18 | P3 | Priority 3 → finish (BT=2) |
| 18–19 | P4 | Priority 4 → finish (BT=1) |

**Step 2: CT, TAT, WT**

| **Process** | **CT** | **TAT=CT−AT** | **WT=TAT−BT** |
|---|---|---|---|
| P2 | 1 | 1 | 0 |
| P5 | 6 | 6 | 1 |
| P1 | 16 | 16 | 6 |
| P3 | 18 | 18 | 16 |
| P4 | 19 | 19 | 18 |

**Step 3: Averages**

- **AWT** = (0+1+6+16+18)/5 = 8.2 ms
- **ATAT** = (1+6+16+18+19)/5 = 12 ms

*In this case, preemptive priority gives the same as non-preemptive because no higher priority process arrives later.*

---

### iv) Round Robin (Quantum = 1 ms)

**Step 1: Timeline using RR (quantum=1)**

- Ready queue order: P1, P2, P3, P4, P5
- Each executes 1 ms, then next process

Execution order (simplified):

- t=0–1 → P1 (remaining 9)
- t=1–2 → P2 (remaining 0 → finish)
- t=2–3 → P3 (remaining 1)
- t=3–4 → P4 (remaining 0 → finish)
- t=4–5 → P5 (remaining 4)
- t=5–6 → P1 (remaining 8)
- Continue in round-robin until all finish

**Step 2: Calculate approximate CT**

| **Process** | **CT** |
|---|---|
| P1 | 20 |
| P2 | 2 |

| Process | CT |
|---|---|
| P3 | 7 |
| P4 | 4 |
| P5 | 17 |

### Step 3: TAT and WT

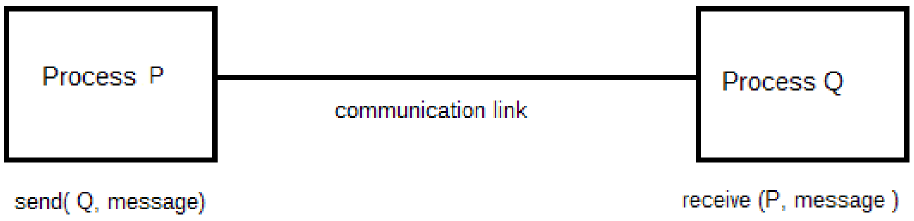
| Process | BT | TAT=CT−AT | WT=TAT−BT |
|---|---|---|---|
| P1 | 10 | 20 | 10 |
| P2 | 1 | 2 | 1 |
| P3 | 2 | 7 | 5 |
| P4 | 1 | 4 | 3 |
| P5 | 5 | 17 | 12 |

### Step 4: Averages

- **AWT** = (10+1+5+3+12)/5 = 6.2 ms
- **ATAT** = (20+2+7+4+17)/5 = 10 ms

---

## Q.3 (a) Define and explain Message Passing.



## Definition

**Message passing** is a method of communication used in operating systems where **processes exchange information** by sending and receiving messages. It is commonly used in **inter-**

**process communication (IPC)**, especially in systems where processes do **not share memory**.

---

## Key Points

1. Processes communicate by explicitly sending and receiving messages.
2. Messages can contain data, commands, or status information.
3. Message passing is used in **distributed systems** and systems with separate address spaces.

---

## Mechanism

Message passing involves two basic operations:

1. **Send(Message)**
   - A process sends a message to another process.
   - Syntax: `send(destination, message)`
2. **Receive(Message)**
   - A process receives a message from another process.
   - Syntax: `receive(source, message)`

---

## Types of Communication

1. **Synchronous (Blocking) Communication**
   - Sender waits until the receiver has received the message.
   - Receiver waits until a message arrives.
2. **Asynchronous (Non-blocking) Communication**
   - Sender sends the message and continues execution immediately.
   - Receiver can receive the message later.

---

## Advantages

- Useful in **distributed systems** where shared memory is not feasible.
- Simplifies process interaction and coordination.
- Provides **synchronization** in blocking mode.

---

## Disadvantages

- Overhead due to message transmission.

- Potential for **deadlock** if processes wait indefinitely.
- Slower than shared memory communication for large data.

(b) Provide a brief description of the following: 04 2 i) Starvation ii) Aging

### Q.3 (b) Brief Description

---

#### i) Starvation

**Starvation** occurs when a process is **indefinitely delayed** from getting CPU or required resources while other processes continue to execute. This usually happens in **priority-based scheduling systems**, where higher-priority processes keep arriving and continuously preempt lower-priority processes. As a result, low-priority processes may never get a chance to execute, leading to unfairness and inefficient resource utilization. Starvation can affect system performance and may cause some processes to remain in the ready queue for a very long time.

---

#### ii) Aging

**Aging** is a technique used by operating systems to **prevent starvation** and ensure fairness. In aging, the priority of a process is **gradually increased the longer it waits** in the ready queue. Over time, even low-priority processes will gain higher priority, eventually allowing them to access the CPU. Aging is commonly applied in **priority scheduling algorithms** to avoid situations where some processes are permanently delayed due to continuously arriving higher-priority processes. This mechanism improves overall system fairness and prevents indefinite waiting.

(c) What is a scheduler? Illustrate queuing with an appropriate diagram.

### Scheduler:

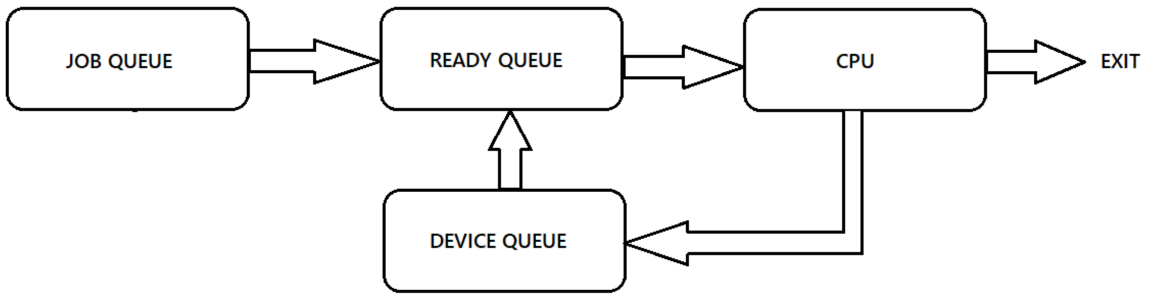
A **scheduler** is a component of the operating system responsible for **selecting which process will run next on the CPU**. The scheduler manages **process queues** and allocates CPU time according to specific scheduling algorithms.

Schedulers ensure **efficient CPU utilization**, **fairness**, and **process coordination** in a multitasking environment.

### Types of Schedulers

1. **Long-term scheduler (Job Scheduler)**
   - Selects processes from the **job pool** (secondary memory) to load into main memory.

    - Controls **degree of multiprogramming**.
2. **Medium-term scheduler**
    - Temporarily **removes processes from main memory** (suspension) and later resumes them.
    - Helps balance CPU and I/O load.
3. **Short-term scheduler (CPU Scheduler)**
    - Selects a process from the **ready queue** to execute on the CPU next.
    - Runs frequently (every few milliseconds).



## Components in the Diagram

1. **Job Queue**
    - Contains **all processes in the system**, whether they are in memory or waiting to be loaded.
    - Every process created by the user or system enters this queue first.
2. **Ready Queue**
    - Contains processes that are **in main memory and ready to execute**, waiting for CPU allocation.
    - Managed by the **short-term scheduler** (CPU scheduler).
3. **CPU**
    - Executes the processes taken from the **ready queue**.
    - The process remains in the CPU until it **completes execution** or is **interrupted** (e.g., for I/O or preemption).
4. **Device Queue**
    - Contains processes **waiting for I/O devices** (e.g., printer, disk).
    - When a process requests I/O, it moves from **CPU → Device Queue**.
    - After the I/O operation completes, it moves back to the **Ready Queue**.
5. **Exit**
    - Processes that **complete execution** in the CPU move to the exit state and are removed from the system.

---

## Flow Explanation

1. New processes are added to the **Job Queue**.
2. The **scheduler** selects processes from the job queue and moves them to the **Ready Queue** when memory is available.
3. The **CPU scheduler** selects a process from the **Ready Queue** to execute.
4. If the process needs I/O, it is moved to the **Device Queue**.
5. Once the I/O operation is completed, the process returns from the **Device Queue** → **Ready Queue**.
6. When a process finishes execution, it exits the CPU and moves to **Exit**.

OR

## Q.3 (a) Differentiate the following: i) Semaphore ii) Monitor

| Aspect | Semaphore | Monitor |
|---|---|---|
| **Definition** | A semaphore is a **synchronization variable** used to control access to shared resources by multiple processes. | A monitor is a **high-level synchronization construct** that encapsulates shared data, procedures, and the synchronization mechanism. |
| **Type** | Can be **counting semaphore** or **binary (mutex) semaphore**. | Always provides **mutual exclusion** automatically. |
| **Implementation** | Implemented using **integer counters** and operations `wait(P)` and `signal(V)`. | Implemented as a **programming language construct** with methods and condition variables. |
| **Ease of Use** | Requires careful programming; prone to errors like **deadlock** and **priority inversion**. | Easier to use; the compiler/language handles mutual exclusion. |
| **Mechanism** | Uses **signals** and **flags** to allow or block processes. | Processes **enter methods of the monitor one at a time**, waiting on condition variables if necessary. |
| **Level** | Low-level synchronization primitive. | High-level synchronization mechanism. |
| **Scope** | Only handles **mutual exclusion** and process synchronization. | Encapsulates **data, procedures, and synchronization** together. |
| **Example Usage** | Used in **producer-consumer, reader-writer problems**. | Used in **bounded buffer problem**, easily solving synchronization without explicit semaphore manipulation. |

## (b) Provide a brief description of the following: i) Deadlock Prevention ii) Deadlock Avoidance

**Deadlock Prevention:**

- Deadlock prevention is a method used to ensure that deadlocks cannot occur in a system by carefully designing the way resources are requested and allocated.
- It works by eliminating at least one of the four necessary conditions for deadlock: mutual exclusion, hold and wait, no preemption, and circular wait.
- The system may require processes to request all required resources at the start, preventing them from holding some resources while waiting for others.
- In some cases, resources may be preempted from processes to avoid deadlock situations.
- Mutual exclusion can be restricted for sharable resources so that multiple processes can use them simultaneously.
- By enforcing such rules, the system guarantees that deadlock will never occur under any circumstances.
- This method is proactive and focuses on preventing deadlocks before they happen rather than dealing with them afterward.

**Deadlock Avoidance:**

- Deadlock avoidance is a method that allows more flexible resource allocation but ensures that the system never enters an unsafe state that could lead to deadlock.
- The system carefully examines each resource request and grants it only if doing so keeps the system in a safe state, where a sequence of process execution exists that allows all processes to complete.
- Algorithms like Banker’s Algorithm are commonly used for deadlock avoidance, which consider the maximum resource requirements of all processes and the current allocation before granting requests.
- Unlike prevention, avoidance does not restrict how resources are requested, but it requires the system to continuously monitor resource states.
- This method balances the need to maximize resource utilization with the requirement to avoid deadlocks, allowing processes to proceed safely while using resources efficiently.
- Deadlock avoidance is more flexible and allows higher resource utilization compared to deadlock prevention because it does not impose rigid allocation restrictions.

## (c) Describe Peterson's Algorithm for Process Synchronization.

**Peterson’s Algorithm for Process Synchronization**

Peterson’s Algorithm is a classical software-based solution used in operating systems to achieve **mutual exclusion** between two processes that share a single resource or critical section. It is one of the simplest and earliest algorithms for process synchronization that does not require special hardware instructions. The main goal of the algorithm is to prevent race conditions when two processes attempt to access shared resources simultaneously while ensuring fairness, avoiding deadlock, and preventing starvation.

The algorithm uses **two shared variables**:

1. `flag[2]` – An array used to indicate whether a process wants to enter the critical section. `flag[i] = true` means process `i` is interested in entering the critical section.
2. `turn` – A variable that indicates which process's turn it is to enter the critical section if both processes want access simultaneously.

**Working of Peterson's Algorithm:**

1. When a process wants to enter the critical section, it first sets its own flag to true, indicating its interest: `flag[i] = true`.
2. It then assigns the `turn` variable to the other process (`turn = j`) to give the other process priority if there is a conflict.
3. The process enters a waiting loop (busy waiting) while the other process is interested in entering (`flag[j] == true`) **and** it is the other process's turn (`turn == j`).
4. When the waiting condition becomes false, the process enters the critical section safely. Only one process can enter at a time, ensuring mutual exclusion.
5. After completing its work in the critical section, the process resets its flag to false (`flag[i] = false`), signaling that it no longer needs the resource, allowing the other process to proceed if it was waiting.

**Key Features of Peterson's Algorithm:**

- **Mutual Exclusion:** Only one process can enter the critical section at any given time, preventing simultaneous access to shared resources.
- **Progress:** If no process is in the critical section, a process that wants to enter will eventually be allowed to enter.
- **Bounded Waiting (Fairness):** Each process gets a fair chance to enter the critical section without indefinite postponement, preventing starvation.
- **Simplicity:** It requires only two shared variables and simple logical checks, making it easy to understand and implement in software.

**Limitations and Use:**

- Peterson's Algorithm works only for **two processes**. Extending it to multiple processes is complex.
- It uses **busy waiting**, which wastes CPU cycles while a process waits.
- It is mostly of theoretical importance, as modern systems typically use hardware-based synchronization primitives like semaphores, mutexes, or monitors.

**Peterson's Algorithm (for two processes P0 and P1)**

**Shared Variables:**

```
flag[2]  // flag[i] is true when process i wants to enter critical section
turn     // indicates which process's turn it is
```

**Pseudo-code for Process Pi (i = 0 or 1, j = 1-i):**

```
flag[i] = true;         // Indicate interest in entering critical section
turn = j;               // Give priority to the other process

// Wait until either the other process is not interested or it is our turn
while (flag[j] == true && turn == j)
    ;  // Busy waiting

// Critical Section starts
// Execute code that requires exclusive access

// Critical Section ends
flag[i] = false;        // Indicate we are leaving the critical section
```

**Explanation of Steps:**

1. Process Pi sets its flag to true to indicate it wants to enter the critical section.
2. It sets `turn = j` to allow the other process priority if both want access simultaneously.
3. The process waits (busy wait) while the other process wants to enter and it is the other process's turn.
4. When the condition becomes false, it enters the critical section safely.
5. After finishing, it sets its flag to false to allow the other process to enter if waiting.

## Q.4 (a) Compare Multiprogramming with Fixed Partition and Multiprogramming with Variable Partition.

| Feature | Fixed Partition | Variable Partition |
|---|---|---|
| **Definition** | Memory is divided into fixed-size partitions at system startup. Each partition can hold exactly one process. | Memory is divided dynamically according to the size of the process. Partitions are created as needed. |
| **Memory Utilization** | Can lead to internal fragmentation because a process may not fully use its partition. | More efficient utilization; only unused memory outside partitions is wasted (external fragmentation may occur). |

| Feature | Fixed Partition | Variable Partition |
|---|---|---|
| **Process Size** | Limited by partition size; large processes may not fit into any partition. | Can accommodate processes of varying sizes as partitions are dynamically allocated. |
| **Flexibility** | Low flexibility; fixed partitions cannot be resized at runtime. | High flexibility; partitions can grow or shrink according to process requirements. |
| **Complexity** | Simple to implement; easy memory management. | More complex; requires dynamic allocation and compaction to reduce fragmentation. |
| **Overhead** | Low overhead in managing memory. | Higher overhead due to dynamic allocation and management of variable partitions. |

## (b) Define Authentication. How can operating system security be ensured using authentication?

**Authentication:**

Authentication is the process by which an operating system or a computing system verifies the identity of a user, process, or device attempting to access the system. It is a fundamental security mechanism that ensures that only legitimate and authorized users can access system resources. Authentication typically involves validating credentials provided by the user, which can include passwords, personal identification numbers (PINs), biometric information such as fingerprints or facial recognition, smart cards, security tokens, or digital certificates. By confirming the identity of the user, authentication prevents unauthorized access and protects the system from malicious activities, data breaches, and misuse of resources.

**Ensuring Operating System Security Using Authentication:**

1. **User Login Verification:** The operating system requires users to provide valid credentials before granting access. This process ensures that unauthorized users cannot log into the system or gain access to sensitive data. Each user's identity is verified against stored authentication information.
2. **Access Control and Permissions:** Once a user is authenticated, the operating system assigns access rights and privileges based on the user's identity. Files, applications,

and system resources can only be accessed by users with proper authorization, preventing unauthorized modifications or data leaks.

3. **Multi-Factor Authentication (MFA):** To increase security, operating systems can implement multi-factor authentication, requiring users to provide two or more forms of verification, such as a password and a one-time token sent to a mobile device, or a password combined with biometric verification. This adds an extra layer of protection against unauthorized access.
4. **Accountability and Audit Trails:** Authentication allows the operating system to maintain detailed logs of user activity. By tracking which authenticated user accessed which resources and when, the system can detect suspicious activity, investigate security breaches, and ensure accountability.
5. **Session Management:** After authentication, the operating system manages user sessions securely. Inactivity timeouts, automatic logouts, and session token management help prevent unauthorized access if a user leaves a system unattended.
6. **Integration with Encryption and Security Policies:** Authentication can work alongside encryption and other security mechanisms to protect data in storage and transit. Only authenticated users can decrypt sensitive information, ensuring confidentiality and integrity.

(c) 1. What is Stripping in RAID? Give an appropriate example.
2. Suppose that a disk drive has 200 cylinders. The drive currently serves at cylinder 50. The queue of pending requests in FIFO order is 82, 170, 43, 140, 24, 16, 190. Starting from the current head position, what is the total distance (in cylinders) the disk arm moves to satisfy all the pending requests for each of the following disk scheduling algorithms?
i) FCFS
ii) SCAN
iii) C-SCAN

## 1. What is Striping in RAID?

**Definition:**
Striping is a technique used in RAID (Redundant Array of Independent/Inexpensive Disks) where data is divided into fixed-size blocks (stripes) and distributed evenly across multiple

disks. Each disk stores a portion of the data, allowing simultaneous read/write operations on multiple disks, which improves performance.

**Key Points:**

- Striping **increases data transfer speed** because multiple disks can be read or written in parallel.
- It does **not provide redundancy** by itself (e.g., RAID 0).
- Often combined with mirroring or parity in RAID levels like RAID 5 or RAID 10 to ensure fault tolerance.

**Example:**
Suppose we have 4 disks (D0, D1, D2, D3) and data blocks A, B, C, D, E, F, G, H:

| Block | Disk Placement |
|---|---|
| A | D0 |
| B | D1 |
| C | D2 |
| D | D3 |
| E | D0 |
| F | D1 |
| G | D2 |
| H | D3 |

Here, blocks are striped across all disks. If a read request comes for blocks A–D, all 4 disks can deliver their blocks simultaneously, improving speed.

---

## 2. Disk Scheduling Problem

**Given:**

- Disk has 200 cylinders (0–199)
- Current head position = 50
- Request queue (FIFO order) = 82, 170, 43, 140, 24, 16, 190

We calculate **total distance** (in cylinders) for each scheduling algorithm.

---

### *i) FCFS (First-Come, First-Served)*

FCFS services requests in the order they arrive.

**Calculation:**

- Start at 50 → 82 → distance = |82–50| = 32
- 82 → 170 → distance = |170–82| = 88
- 170 → 43 → distance = |43–170| = 127
- 43 → 140 → distance = |140–43| = 97
- 140 → 24 → distance = |24–140| = 116
- 24 → 16 → distance = |16–24| = 8
- 16 → 190 → distance = |190–16| = 174

**Total Distance:**
32 + 88 + 127 + 97 + 116 + 8 + 174 = **642 cylinders**

---

### *ii) SCAN (Elevator Algorithm)*

- Head moves in one direction (assume toward higher cylinder first), servicing all requests until the end, then reverses.
- Requests sorted: 16, 24, 43, 82, 140, 170, 190
- Current head = 50, moving **up** first: 82, 140, 170, 190 → reach end 199 → reverse → 43, 24, 16

**Distance calculation:**

- 50 → 82 = 32
- 82 → 140 = 58
- 140 → 170 = 30
- 170 → 190 = 20
- 190 → 199 = 9 (to end)
- Reverse: 199 → 43 = 156
- 43 → 24 = 19
- 24 → 16 = 8

**Total Distance:**
32 + 58 + 30 + 20 + 9 + 156 + 19 + 8 = **332 cylinders**

---

### *iii) C-SCAN (Circular SCAN)*

- Head moves in one direction only (up toward higher cylinders), servicing requests.
- When it reaches the end, it **jumps to the beginning** without servicing requests, then continues.
- Requests sorted: 16, 24, 43, 82, 140, 170, 190
- Current head = 50, moving **up**: 82, 140, 170, 190 → reach end 199 → jump to 0 → service 16, 24, 43

**Distance calculation:**

- 50 → 82 = 32
- 82 → 140 = 58
- 140 → 170 = 30
- 170 → 190 = 20
- 190 → 199 = 9
- Jump: 199 → 0 = 199
- 0 → 16 = 16
- 16 → 24 = 8
- 24 → 43 = 19

**Total Distance:**
32 + 58 + 30 + 20 + 9 + 199 + 16 + 8 + 19 = **391 cylinders**

---

OR

Q.4 (a) What is TLB(Translation Lookaside Buffer), and what is the purpose of it?

**Translation Lookaside Buffer (TLB):**

A **Translation Lookaside Buffer (TLB)** is a special, small, and fast **cache memory** used by the CPU to improve the speed of virtual-to-physical address translation in a paging-based memory management system. In a virtual memory system, each process generates **virtual addresses** that need to be translated into **physical addresses** in main memory using the page table. Accessing the page table in main memory for every memory reference can be slow. The TLB stores **recently used page table entries** so that the CPU can quickly find the physical address without accessing main memory.

**Purpose of TLB:**

1. **Speed up memory access:** By caching recent virtual-to-physical address mappings, the TLB reduces the time needed for address translation, improving CPU performance.
2. **Reduce memory access overhead:** Without a TLB, every memory reference would require accessing the page table in main memory, which is slower.
3. **Efficient virtual memory management:** The TLB allows the system to handle large virtual address spaces efficiently while maintaining high-speed memory access for frequently used pages.

4. **Support for multiple processes:** Modern CPUs maintain TLB entries with process identifiers (tags) so that multiple processes can share the TLB without conflicts.

**Working Example:**

- Suppose a process wants to access virtual address `0x1234`.
- The CPU first checks the TLB to see if the page containing `0x1234` has a valid physical address mapping.
  - If it is found (**TLB hit**), the CPU can immediately access the physical memory.
  - If it is not found (**TLB miss**), the system looks up the page table in main memory, retrieves the physical address, updates the TLB with this mapping, and then accesses memory.

The TLB significantly **reduces the average memory access time** in systems using virtual memory, making it a crucial component of modern CPU architecture.

## (b) Discuss the Access Control List in brief.

**Access Control List (ACL):**

An **Access Control List (ACL)** is a security mechanism implemented in operating systems to manage and enforce **permissions on system resources**. It defines **which users or groups (subjects) can access a particular resource** and specifies the type of operations they are allowed to perform. ACLs are associated with every resource such as files, directories, devices, or network resources, and act as a detailed permission table that the operating system references whenever a user or process attempts to access that resource.

**Key Features and Functioning:**

1. **Fine-Grained Access Control:** Unlike simple permission systems, ACLs allow different users or groups to have **different levels of access** for the same resource. This enables highly customized security policies.

2. **Subject-Object Mapping:** Each entry in an ACL maps a **subject** (user or group) to a set of allowed **operations** (read, write, execute, delete, etc.) on the **object** (the resource).
3. **Enforcement:** When a user tries to access a resource, the operating system consults the ACL of that resource. If the requested operation is permitted for the user or group, access is granted; otherwise, it is denied.
4. **Hierarchical and Group Permissions:** ACLs can include both individual users and groups. This allows system administrators to assign permissions to groups, simplifying management for large numbers of users.
5. **Dynamic Management:** ACLs can be updated at runtime, allowing administrators to add or remove users, change permissions, or revoke access without affecting other system operations.

**Example:**
Consider a file `project.docx` with the following ACL:

| **User/Group** | **Permissions** |
|---|---|
| Alice | Read, Write |
| Bob | Read |
| Manager | Read, Write, Execute |
| Guest | None |

- Alice can read and modify the file but cannot execute any scripts.
- Bob can only view the file.
- Manager has full access, including execution rights.
- Guest has no access at all.

**Purpose and Advantages:**

- ACLs enhance **security** by ensuring that only authorized users can access or modify resources.
- They provide **accountability**, as each action can be traced back to a specific user.
- ACLs help maintain **data integrity and confidentiality**, preventing accidental or malicious alterations.

- They are essential for **multi-user systems**, where different users require different levels of access to shared resources.

In modern operating systems, ACLs are widely used not just for files and directories but also for **network resources, devices, and system services**, forming a critical part of the overall security infrastructure.

(c) 1. What is Mirroring in RAID? Give an appropriate example.

2. Suppose that a disk drive has 200 cylinders. The drive currently

serves at cylinder 100; the previous request was at cylinder 81. The

queue of pending requests in FIFO order is

55,58,39,18,90,160,150,38,184. Starting from the current head

position, what is the total distance (in cylinders) the disk arm

moves to satisfy all the pending requests for each of the following

disk scheduling algorithms?

i) SSTF

ii) LOOK

iii) C-LOOK

**1. Mirroring in RAID**

**Definition:**
Mirroring in RAID is a technique where data is **duplicated identically on two or more disks**. Each disk in the mirror contains the exact same copy of the data. This provides **fault tolerance**, as if one disk fails, the system can continue operating using the copy from the other disk(s).

**Key Points:**

- Provides **high reliability** because every piece of data exists on at least two disks.
- Write operations are slower than read operations because data must be written to all mirrored disks.
- Commonly used in **RAID 1**.

**Example:**
Suppose we have two disks, D0 and D1, and we want to store blocks A, B, C, D:

| Block | Disk D0 | Disk D1 |
|---|---|---|
| A | A | A |
| B | B | B |
| C | C | C |
| D | D | D |

If disk D0 fails, all data is still available on D1, ensuring no data loss.

---

**2. Disk Scheduling Problem**

**Given:**

- Disk cylinders: 0–199
- Current head: 100
- Previous request: 81 (indicating head movement direction toward lower cylinders)
- Request queue (FIFO order): 55, 58, 39, 18, 90, 160, 150, 38, 184

We calculate **total distance** for each scheduling algorithm.

---

## i) SSTF (Shortest Seek Time First)

SSTF services the request closest to the current head position at each step.

**Initial head:** 100

**Step-by-step servicing (closest first):**

1. 100 → 90 (closest) = 10
2. 90 → 58 = 32
3. 58 → 55 = 3
4. 55 → 39 = 16
5. 39 → 38 = 1
6. 38 → 18 = 20
7. 18 → 150 (next closest from remaining: 150, 160, 184) = 132
8. 150 → 160 = 10
9. 160 → 184 = 24

**Total head movement:** 10 + 32 + 3 + 16 + 1 + 20 + 132 + 10 + 24 = **248 cylinders**

---

## ii) LOOK

LOOK moves the head in the direction of pending requests and reverses at the last request in that direction instead of going to the disk end. Since the previous request was 81, head moves **toward lower cylinders first**.

**Requests sorted:** 18, 38, 39, 55, 58, 90, 150, 160, 184

**Step-by-step:**

- Moving downward first: 100 → 90 → 58 → 55 → 39 → 38 → 18
- Then reverse upward to remaining: 18 → 150 → 160 → 184

**Calculations:**

1. 100 → 90 = 10
2. 90 → 58 = 32
3. 58 → 55 = 3
4. 55 → 39 = 16
5. 39 → 38 = 1
6. 38 → 18 = 20
7. 18 → 150 = 132
8. 150 → 160 = 10
9. 160 → 184 = 24

**Total head movement:** 10 + 32 + 3 + 16 + 1 + 20 + 132 + 10 + 24 = **248 cylinders**

*(In this case, SSTF and LOOK give the same total distance because of request distribution.)*

---

## iii) C-LOOK (Circular LOOK)

C-LOOK moves the head in one direction only, servicing requests, and jumps back to the **lowest request** after reaching the highest. Head moves **toward lower cylinders first**, then jumps to the smallest request in the upward direction.

**Sorted requests:** 18, 38, 39, 55, 58, 90, 150, 160, 184

**Step-by-step:**

- Head downward: 100 → 90 → 58 → 55 → 39 → 38 → 18
- Jump from lowest request 18 → next request in upward direction 150 → 160 → 184

**Distance calculation:**

1. 100 → 90 = 10

2. 90 → 58 = 32
3. 58 → 55 = 3
4. 55 → 39 = 16
5. 39 → 38 = 1
6. 38 → 18 = 20
7. Jump: 18 → 150 = 132
8. 150 → 160 = 10
9. 160 → 184 = 24

**Total head movement:** 10 + 32 + 3 + 16 + 1 + 20 + 132 + 10 + 24 = **248 cylinders**

---

**Observation:**
In this particular request pattern, SSTF, LOOK, and C-LOOK result in the **same total head movement = 248 cylinders** due to the distribution of requests and head movement direction.

## Q.5 (a) Enlist the advantages and disadvantages of Virtual Machines.

**Advantages:**

1. **Hardware Independence:** VMs allow operating systems and applications to run on any physical hardware, as the virtual machine abstracts the underlying hardware.
2. **Isolation:** Each VM runs in a separate environment, so failures or crashes in one VM do not affect other VMs or the host system.
3. **Resource Consolidation:** Multiple VMs can run on a single physical machine, improving hardware utilization and reducing costs.
4. **Easy Backup and Recovery:** VMs can be easily cloned, backed up, or restored, simplifying system maintenance and disaster recovery.
5. **Testing and Development:** VMs provide a safe environment for testing new software, patches, or operating systems without affecting the host system.
6. **Security:** VMs can be used to sandbox untrusted applications, reducing the risk of malware affecting the host system.
7. **Portability:** VMs can be moved or migrated between physical machines without modification, enabling flexible deployment.

**Disadvantages:**

1. **Performance Overhead:** Running multiple VMs consumes CPU, memory, and storage resources, which may reduce performance compared to running directly on physical hardware.

2. **Resource Contention:** Multiple VMs sharing the same physical resources can lead to competition, causing slower performance or delays.
3. **Complex Management:** Managing and maintaining multiple VMs, including updates, security, and monitoring, can be complex.
4. **Licensing Costs:** Using virtualization software or running multiple OS instances may incur additional licensing expenses.
5. **Limited Hardware Access:** Some applications that require direct access to specific hardware may not run efficiently or at all in a VM.
6. **Security Risks in VM Escape:** If a vulnerability allows a VM to break out of its isolated environment, it could compromise the host system or other VMs.

Virtual machines provide flexibility, isolation, and efficiency but must be carefully managed to balance performance, security, and resource utilization.

(b) Write a shell script to display all executable files, directories, and zero sized files from the current directory.

```
#!/bin/bash

# Display all executable files
echo "Executable files in the current directory:"
for file in *; do
    if [ -f "$file" ] && [ -x "$file" ]; then
        echo "$file"
    fi
done

echo ""

# Display all directories
echo "Directories in the current directory:"
for dir in *; do
```

```
    if [ -d "$dir" ]; then
        echo "$dir"
    fi
done

echo ""

# Display all zero-sized files
echo "Zero-sized files in the current directory:"
for file in *; do
    if [ -f "$file" ] && [ ! -s "$file" ]; then
        echo "$file"
    fi
done
```

After running the shell script, the **output will be**:

```
Executable files in the current directory:
script.sh
program

Directories in the current directory:
docs
images

Zero-sized files in the current directory:
empty.txt
```

(c) Consider the reference string: 4, 7, 6, 1, 7, 6, 1, 2, 7, 2. The number of frames in the memory is 3. Find out the number of page faults respective to:

i) Optimal Page Replacement Algorithm

ii) FIFO Page Replacement Algorithm

iii) LRU Page Replacement Algorithm

We are given:

- **Reference string:** 4, 7, 6, 1, 7, 6, 1, 2, 7, 2
- **Number of frames:** 3

We need to find the **number of page faults** for different page replacement algorithms. Let's solve **step by step**.

---

## i) Optimal Page Replacement Algorithm

**Rules:** Replace the page that will not be used for the **longest time in the future**.

**Step-by-step:**

1. **Frames empty initially** → 4 → page fault → [4]
2. 7 → page fault → [4,7]
3. 6 → page fault → [4,7,6]
4. 1 → page fault → choose page to replace:
   - Future use: 4(not used again), 7(used at 5), 6(used at 6) → replace 4 → [1,7,6]
5. 7 → already in frame → no page fault
6. 6 → already in frame → no page fault
7. 1 → already in frame → no page fault
8. 2 → page fault → choose page to replace:
   - Frames: 1,7,6
   - Future use: 1(not used again), 7(used at 9), 6(not used again) → replace 1 or 6 (both not used again) → replace 1 → [2,7,6]
9. 7 → already in frame → no page fault
10. 2 → already in frame → no page fault

**Total page faults (Optimal): 6**

---

## ii) FIFO (First-In, First-Out) Page Replacement Algorithm

**Rules:** Replace the **oldest page** in memory.

**Step-by-step:**

1. 4 → page fault → [4]
2. 7 → page fault → [4,7]
3. 6 → page fault → [4,7,6]
4. 1 → page fault → replace 4 (first in) → [1,7,6]
5. 7 → already in frame → no page fault
6. 6 → already in frame → no page fault
7. 1 → already in frame → no page fault
8. 2 → page fault → replace 7 (next FIFO) → [1,2,6]
9. 7 → page fault → replace 6 → [1,2,7]

10. 2 → already in frame → no page fault

**Total page faults (FIFO): 7**

---

### iii) LRU (Least Recently Used) Page Replacement Algorithm

**Rules:** Replace the page that was **least recently used**.

**Step-by-step:**

1. 4 → page fault → [4]
2. 7 → page fault → [4,7]
3. 6 → page fault → [4,7,6]
4. 1 → page fault → LRU = 4 (used at step 1, least recently used) → [1,7,6]
5. 7 → already in frame → update recent usage → [1,7,6]
6. 6 → already in frame → update recent usage → [1,7,6]
7. 1 → already in frame → update recent usage → [1,7,6]
8. 2 → page fault → LRU = 7 (used at step 5, least recently used among 1,7,6) → [1,2,6]
9. 7 → page fault → LRU = 6 (used at step 6) → [1,2,7]
10. 2 → already in frame → update recent usage → [1,2,7]

**Total page faults (LRU): 7**

---

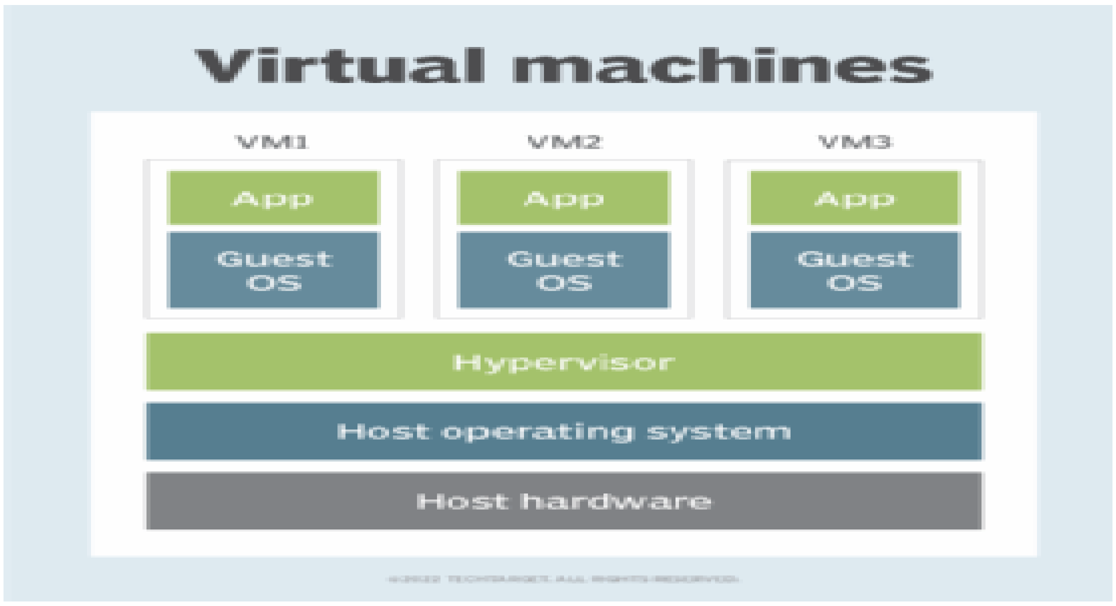
**Answer:**

| Algorithm | Page Faults |
|---|---|
| Optimal | 6 |
| FIFO | 7 |
| LRU | 7 |

OR

## Q.5 (a) Describe virtualization with an appropriate diagram

Virtualization is the process of creating **virtual instances of computer resources**, such as hardware, operating systems, storage, or networks, allowing multiple independent systems to run on a single physical machine. It enables better **resource utilization, flexibility, isolation, and management** of computing resources.



## 1. Host Hardware Layer

The **host hardware** is the foundation of the virtualization environment. It includes all physical components of the computer such as:

- **CPU (Central Processing Unit):** Performs all processing tasks and executes instructions for both the host OS and virtual machines.
- **Memory (RAM):** Stores active programs, data, and virtual machine states temporarily for quick access.
- **Storage Devices:** Hard disks or SSDs where the host OS, guest OS, and applications store persistent data.
- **Network Interfaces:** Enable connectivity to internal or external networks for the host system and VMs.

This layer provides the **physical resources** that are abstracted and shared among virtual machines by the hypervisor. Efficient hardware resources are crucial for supporting multiple VMs simultaneously without performance degradation.

---

## 2. Host Operating System (Host OS) Layer

The **host OS** runs directly on the physical hardware and provides the **basic operating environment** for the system. It manages the hardware resources, schedules processes, handles I/O operations, and provides services such as file systems and network communication.

In virtualization:

- The host OS allows the hypervisor to interact with the hardware.
- It ensures that the hypervisor has secure and controlled access to CPU cycles, memory allocation, and storage.

- Some virtualization approaches (like Type 2 hypervisors) rely on the host OS for device drivers and I/O management.

Essentially, the host OS acts as the **bridge between physical hardware and the virtualization software**, enabling VMs to operate seamlessly.

---

## 3. Hypervisor Layer

The **hypervisor**, or virtual machine monitor (VMM), is the core component of virtualization. Its main responsibilities are:

- **Creating and Managing VMs:** Instantiates virtual machines, allocates CPU, memory, and storage, and starts/stops VMs.
- **Resource Allocation:** Dynamically distributes hardware resources among VMs based on demand and priority.
- **Isolation:** Ensures that each VM operates independently, so failures, crashes, or security breaches in one VM do not affect others.
- **Abstraction:** Presents virtualized hardware to each VM, making them believe they are running on separate physical machines.

---

## 4. Virtual Machine (VM) Layer

Each **virtual machine** is a complete, self-contained system that includes:

- **Guest Operating System (Guest OS):** Can be the same as or different from the host OS, providing the environment for applications to run.
- **Applications (Apps):** Programs installed within the VM that operate as if they were on a physical machine.

VMs are **isolated from each other**, meaning:

- Applications in one VM cannot interfere with another VM.
- Each VM can run different OS versions or even different types of OS (Windows, Linux, etc.) on the same host hardware.

The VM layer allows **flexibility, testing, development, and secure sandboxing**, as multiple environments can coexist without conflicts.

## (b) Advantages of LINUX/UNIX operating system over Windows.

---

1. **Cost-effective:**
   Linux/Unix is mostly free and open-source, so it can be used without paying any license fees.
   Windows requires purchasing licenses, which can be costly for multiple users or

organizations.
This makes Linux/Unix ideal for individuals, schools, and businesses with limited budgets.
Even enterprise Linux versions are cheaper than Windows with similar support services.

---

2. **Secure:**
   Linux/Unix has a strong security model with strict file permissions and user roles.
   It is less prone to viruses, malware, and unauthorized access compared to Windows.
   Frequent updates and a large open-source community quickly address vulnerabilities.
   This makes Linux/Unix a preferred choice for servers, banks, and sensitive systems.

---

3. **Stable and reliable:**
   Linux/Unix can run continuously for months or even years without crashing.
   It efficiently manages system resources and handles multiple tasks simultaneously.
   Windows often needs frequent reboots after updates or crashes, reducing uptime.
   High stability makes Linux/Unix ideal for servers and mission-critical applications.

---

4. **Customizable:**
   Linux/Unix allows users to modify and configure the operating system according to their needs.
   Users can choose desktop environments, modify the kernel, and install only necessary components.
   This flexibility is useful for creating specialized systems or lightweight installations.
   Customizability makes it popular among developers, enthusiasts, and IT professionals.

---

5. **Developer-friendly:**
   Linux/Unix provides built-in programming tools, compilers, shells, and scripting environments.
   It supports multiple programming languages and open-source software development.
   Developers can easily automate tasks and manage servers through command-line tools.
   This makes Linux/Unix ideal for programmers, web developers, and software engineers.

---

6. **Networking capabilities:**
   Linux/Unix has strong built-in networking support and tools for servers and administration.
   It comes with firewalls, remote access utilities, and server applications by default.

Windows requires additional software and configuration to achieve similar networking features.
This makes Linux/Unix highly suitable for web servers, mail servers, and network management.

---

7. **Performance:**
   Linux/Unix runs efficiently even on older hardware and consumes fewer resources.
   It has faster boot times and better memory management compared to Windows.
   This allows high-performance computing and smooth operation under heavy workloads.
   Users can choose lightweight distributions for specific needs or low-end machines.

---

8. **Community and support:**
   Linux/Unix has a large global community providing documentation, forums, and tutorials.
   Updates, bug fixes, and improvements are often released faster than in proprietary systems.
   Users can access help for free from forums, online guides, or open-source contributors.
   This strong community makes Linux/Unix reliable, constantly improving, and user-friendly.

---

(c) Given memory partition of 100K, 500K, 200K, 300K, and 600K in order. How would each of the First-fit, Best-fit and Worst-fit algorithms place the processes of 212K, 417K, 112K and 426K in order? Which algorithm makes the most efficient use of memory? Show the diagram of memory status in each case.

**Memory partitions (in order):** 100K, 500K, 200K, 300K, 600K
**Processes (in order):** 212K, 417K, 112K, 426K

We need to allocate them using **First-fit, Best-fit, and Worst-fit** and determine which is most efficient.

---

### 1. First-Fit Algorithm

- **Rule:** Allocate the first partition that is large enough for the process.

**Step-by-step:**

1. Process **212K** → fits in **500K** (first partition ≥ 212K)
    - Remaining partitions: 100K, **288K**, 200K, 300K, 600K
2. Process **417K** → fits in **600K** (first partition ≥ 417K)
    - Remaining partitions: 100K, 288K, 200K, 300K, **183K**
3. Process **112K** → fits in **288K** (first partition ≥ 112K)
    - Remaining partitions: 100K, **176K**, 200K, 300K, 183K
4. Process **426K** → no partition large enough → **not allocated**

**Memory Status Diagram (After First-Fit):**

| Partition | Size (K) | Status |
|---|---|---|
| 1 | 100 | Free |
| 2 | 288 | 112K used |
| 3 | 200 | Free |
| 4 | 300 | Free |
| 5 | 183 | 417K used |

---

## 2. Best-Fit Algorithm

- **Rule:** Allocate the **smallest partition that is large enough** for the process.

**Step-by-step:**

1. Process **212K** → fits in **300K** (smallest ≥ 212K)
    - Remaining partitions: 100K, 500K, 200K, **88K**, 600K
2. Process **417K** → fits in **500K** (smallest ≥ 417K)
    - Remaining partitions: 100K, **83K**, 200K, 88K, 600K
3. Process **112K** → fits in **200K** (smallest ≥ 112K)
    - Remaining partitions: 100K, 83K, **88K**, 88K, 600K
4. Process **426K** → fits in **600K** (smallest ≥ 426K)
    - Remaining partitions: 100K, 83K, 88K, 88K, **174K**

**Memory Status Diagram (After Best-Fit):**

| Partition | Size (K) | Status |
|---|---|---|
| 1 | 100 | Free |
| 2 | 500 | 417K used |
| 3 | 200 | 112K used |
| 4 | 300 | 212K used |

| Partition | Size (K) | Status |
|---|---|---|
| 5 | 600 | 426K used |

## 3. Worst-Fit Algorithm

- **Rule:** Allocate the **largest partition available** for the process.

**Step-by-step:**

1. Process **212K** → fits in **600K** (largest)
   - Remaining partitions: 100K, 500K, 200K, 300K, **388K**
2. Process **417K** → fits in **500K** (largest remaining ≥ 417K)
   - Remaining partitions: 100K, **83K**, 200K, 300K, 388K
3. Process **112K** → fits in **388K** (largest remaining ≥ 112K)
   - Remaining partitions: 100K, 83K, 200K, 300K, **276K**
4. Process **426K** → no partition large enough → **not allocated**

**Memory Status Diagram (After Worst-Fit):**

| Partition | Size (K) | Status |
|---|---|---|
| 1 | 100 | Free |
| 2 | 500 | 417K used |
| 3 | 200 | Free |
| 4 | 300 | Free |
| 5 | 388 | 112K used |

## 4. Efficiency Comparison

- **First-Fit:** 3 processes allocated, 1 unallocated.
- **Best-Fit:** All 4 processes allocated → **most efficient use of memory**.
- **Worst-Fit:** 3 processes allocated, 1 unallocated.

✓ **Best-Fit is the most memory-efficient in this case.**